Лабораторная работа № 1	Группа 05	2023
Представление чисел	Ратьков Андрей Игоревич	

1 Инструментарий и требования к работе

Задача: Необходимо написать программу, которая позволяет выполнять арифметические действия с дробными числами в форматах с фиксированной и плавающей точкой. Программа должна использовать только целочисленные вычисления и типы данных.
Программа написана на Python (3.11.5). Выполнено в среде разработки PyCharm.

2 Ссылка на репозиторий

`https://github.com/skkv-mkn/mkn-comp-arch-2023-fixed-floating-AndrewRatkov`

3 Результат работы на тестовых данных

На моих тестах программа работает (см. главу 4.6). Вот табличка с примером работы каких-то тестов:

Input	Output
h 2 0x7800 + 0x0001	0x1.004p+15
h 3 0x0002 + 0x0001	0x1.800p-23
8.8 3 0xffff * 0x0f00	-0.059
f 1 0x414587dd / 0x42ebf110	0x1.aca5aep-4
h 1 0x0000 + 0xff00	nan
h 0 0x2400 / 0x2400	0x1.000p+0
h 2 0x5401 / 0x4400	0x1.004p+4
h 1 0x7800 + 0x7801	inf
16.16 1 0x173600	23.211
12.8 0 0x038e9 / 0x47eeb	0.046
12.8 1 0x038e9 / 0x47eeb	0.051
12.8 2 0x038e9 / 0x47eeb	0.051
12.8 3 0x038e9 / 0x47eeb	0.046
f 3 0x80000000 + 0x00000000	-0x0.000000p+0
f 3 0x00000000 + 0x80000000	-0x0.000000p+0
f 3 0xdfa2562d + 0x5fa2562d	-0x0.000000p+0
f 3 0x11111111 + 0x91111111	-0x0.000000p+0
h 0 0x1400 * 0x0200	0x0.000p+0
h 1 0x1400 * 0x0200	0x0.000p+0
h 2 0x1400 * 0x0200	0x1.000p-24
h 3 0x1400 * 0x0200	0x0.000p+0
f 0 0x414587dd + 0x42ebf110	0x1.04a20ap+7
f 1 0x414587dd + 0x42ebf110	0x1.04a20cp+7
f 2 0x414587dd + 0x42ebf110	0x1.04a20cp+7
f 3 0x414587dd + 0x42ebf110	0x1.04a20ap+7

4 Как работает код?

Проект состоит из нескольких файлов: main.py, constants.py, basic_functions.py, fixed_point_class.py, floating_point_class.py, tester.sh .

4.1 main.py

Главный файл, который запускается командой типа "python3 main.py [аргументы]" или "python3.11 main.py [аргументы]". Содержит единственную функцию solve, которой на вход подаётся команда пользователя. Если запрос некорректен, функция напечатает, почему она так считает. Если корректен — будет вызвано(-ы) соответвующее(-ие) действие над числами с плавающей или фиксированной точкой.

4.2 constants.py

Тут лежат нужные для работы константы:

- LOCAL_DEBUG — для локального дебага — чтобы запускались assert'ы в разных функциях (когда значение равно True) — было полезно при написании кода
- chars16_to_2 — словарь, переводящий символ 16-ричной системы счисления в список из 4 битов
- hex_string — последовательно все цифры 16-ричной системы счисления в возрастающем порядке и верхнем регистре.
- Rounding_TOWARD_ZERO, ... Rounding_TOWARD_NEG_INFINITY — типы округления, которым соответствуют числа от 0 до 3
- Types_USUAL, Types_PLUS_INF, Types_MINUS_INF, Types_NaN — типы чисел с плавающей точкой — обычное число / бесконечности / NaN'ы — им тоже соответствуют свои числа от 0 до 3

4.3 basic_functions.py

Тут базовые функции, которые понадобятся для всяких махинаций между двоичными записями и числами int (переводов одного в другое и т.п.)

- hex2binary_code(code16) — перевод шестнадцатеричной записи code16 в бинарный код (список из 0 и 1), который получится в 4 раза длиннее
- int2code2(x, bits) — перевод целого числа в код с дополнением до 2 длины bits — нужно в числах с фиксированной точкой.
- int2bin(x, bits) — список из последних bits битов целого числа x.
- bin_code2int(code) — перевод обычного бинарного кода в целое число(int)
- string_can_be_converted_to_int(string) — True/False в зависимости от того, можно ли перевести строку в int (то есть она состоит только из цифр)
- least_bits_needed(x) — минимальное число битов для x, то есть наименьшее L такое что $2^L > x$

4.4 fixed_point_class.py

Числа с фиксированной точкой — объекты класса FixedPointNum. Для дробного числа с фиксированной точкой типа $A.B$ в этом классе хранятся следующие поля:

- self.a — количество битов перед точкой (A)
- self.b — количество битов после точки (B)
- self.code — list из $A+B$ нулей и единиц — код дробного числа (код с добавлением до 2)
- self.rounding — тип округления (целое число от 0 до 3)
- self.integer — целое число, в 2^B раз превосходящее данное (Равно $2^0 \cdot$ [последний бит] $+ 2^1 \cdot$ [предпоследний бит] $+ \ldots + 2^{A+B-2} \cdot$ [второй бит] $- 2^{A+B-1} \cdot$ [первый бит]).

Метод	Описание
__init__	Создание объекта класса по параметрам A, B, коду code и типу округления rounding
__add__(other)	Сложение self + other (складываем соответствующие self.integer и other.integer и переводим сумму к формату $A.B$): Мы хотим сложить два числа, равные $self.integer \cdot 2^{-B}$ и $other.integer \cdot 2^{-B}$. Их сумма будет равна $(self.integer + other.integer) \cdot 2^{-B}$. Нужно помнить, что числа с фиксированной точкой складываются как в модульной арифметике и поэтому, если $(self.integer + other.integer) >= 2^{A+B-1}$, нужно вывести $(self.integer + other.integer - 2^{A+B-1}) \cdot 2^{-B}$, а если $(self.integer + other.integer) < -2^{A+B-1}$, то нужно вывести $(self.integer + other.integer + 2^{A+B-1}) \cdot 2^{-B}$
__sub__(other)	Вычитание self – other (аналогично предыдущему пункту, только + меняем на –): Мы хотим получить разность двух чисел, равных $self.integer \cdot 2^{-B}$ и $other.integer \cdot 2^{-B}$. Их разность будет равна $(self.integer - other.integer) \cdot 2^{-B}$. Нужно помнить, что числа с фиксированной точкой складываются (и, естественно, вычитаются) как в модульной арифметике и поэтому, если $(self.integer - other.integer) >= 2^{A+B-1}$, нужно вывести $(self.integer - other.integer - 2^{A+B-1}) \cdot 2^{-B}$, а если $(self.integer - other.integer) < -2^{A+B-1}$, то нужно вывести $(self.integer - other.integer + 2^{A+B-1}) \cdot 2^{-B}$

__mul__(other)	Умножение self · other (умножаем соответствующие self.integer и other.integer и делим произведение на 2^B (сдвиг вправо на B бит), после чего переводим к формату $A.B$, округляя нужным нам способом) Итак, мы умножаем числа $self.integer \cdot 2^{-B}$ и $other.integer \cdot 2^{-B}$ в нашей модульной арифметике. По факту, мы хотим получить $(self.integer \cdot other.integer \cdot 2^{-B}) \cdot 2^{-B}$. Так и получим это. Перемножим сначала числа $self.integer \cdot other.integer$. Далее нужно отбросить от полученного числа последние B битов (умножение на 2^{-B}) в соответствии с правилами округления: 0. Округление к 0 - просто битовый сдвиг на B битов вправо. 1. Округление к ближайшему чётному — смотрим на последние B битов. Если они (при переводе из двоичной в десятичную) образуют число большее чем 2^{B-1}, то после того, как мы их отбросим, нужно прибавить 1 бит к полученному числу (что будет соответствовать к округлению к ближайшему четному в случае, когда ближайшее число больше нашего). Если последние B битов образуют ровно 2^{B-1}, отбрасываем последние B битов и смотрим на последний бит полученного числа. Если он 0 — ничего не делаем, если 1 — прибавляем 1 бит к числу. Это случай, когда ближайшего нет и мы округляем к чётному. В противном случае мы просто отбрасываем B битов. 2. Округление к плюс бесконечности — воспользуемся тем, что целочисленное деление питоне делит с округлением к минус бесконечности. Умножим число на –1, нацело поделим на 2^B, умножим снова на –1. 3. Округление к минус бесконечности — просто используем питоновское целочисленное деление (оно с округлением к минус бесконечности), делим как и раньше на 2^B. После отбрасывания B правых битов наше число по-прежнему модет на влезать в $A+B$ битов. Поскольку у нас всё по модулю 2^{A+B}, нас не бедет интересовать все биты, кроме $A+B$ младших, которые мы отбросим и приведём число в нужный диапазон.

__truediv__(other)	Деление self/other. Сначала разбираются частные члучаи, когда кто-то 0. В общем случае мы делим self.integer $\cdot\, 2^B$ на other.integer (делимое мы домножаем на 2^B, чтобы получить информацию о битах после точки после деления), после чего нужным способом округляем и переводим в формат $A.B$: Каждый раз мы запоминаем, какого знака должно получиться реальное частное этих чисел. (–, если числа разного знака; + если одинакового), и дальше делим модули чисел. В округлении к 0 или какой-либо бесконечности просто правильно пользуемся питоновским целочисленным делением (которое с округлением к минус бесконечности). В округлении к ближайшему чётному – просто дополнительно смотрим на остаток от деления $self.integer \cdot 2^B$ на $other.integer$. Если он больше половины $other.integer$ или равен ей и частное нечётно, то частное увеличиваем на 1. В конце если частное вышло за диапазон, нужно его вернуть в него (так как мы живём в кольце по модулю 2^{A+B}). В итоге ответ — частное, умноженное на 2^B.
is_neg	True, если число отрицательное; False, если положительное
__str__	Для начала определяем целые переменные $first\_part$, $second\_part$ и $first\_nulls$. $first\_part$ — это ближайшее целое число к нашему числу x, которое находится между x и 0 (округлённое вверх число x (если оно отрицательное), иначе просто целая часть числа). $second\_part$ — целое число, которое образуют цифры десятичной записи после запятой. $first\_nulls$ — количество нулей после запятой до начала $second\_part$. То есть число x в десятичной записи выглядит как {(минус, если число x отрицательное, а $first\_part = 0$) ($first\_part$) . (0 ...0) ($second\_part$)} Например, если десятичная запись числа это -0,00482, то $first\_part = 0, first\_nulls = 2, second\_part = 482$, а если десятичная запись числа это -3,05345, то $first\_part = -3, first\_nulls = 1, second\_part = 5345$. В переменной $rounded\_second\_part$ будем хранить первые три цифры записи числа после запятой. По факту, от нас хотят, чтобы мы выводили числа в формате {целая часть}.{$rounded\_second\_part$}, но $rounded\_second\_part$ иногда приходится менять (увеличивать/уменьшать на 1) в зависимости от правила округления и цифр, которые идут дальше после первых трёх после запятой. При этом, если у нас $rounded\_second\_part = 999$ и её надо увеличить на 1, то она должна стать 000, но надо менять $first\_part$. Это делают подфункции *increase* и *decrease* в функции __str__.

4.5 floating_point_class.py

Числа с плавающей точкой — объекты класса FloatNum. Для них мы храним следующие поля:

- self.half_or_float — True(если это float), False(если это half) В зависимоти от того, half это или float определяются следующие константы:

Поле	Значение для чисел half	Значение для чисел float	Что обозначает
self.EXP_LEN	5	8	Длина экспоненты
self.MANT_LEN	10	23	Длина мантиссы
self.MAX_EXP	16	128	Максимальная экспонента
self.MIN_EXP	-14	-126	Минимальная экспонента у **недермонализованого** числа
self.LEN	16	32	Сколько всего битов отводится под число
self.EXP_SHIFT	15	127	Сдвиг экспоненты при переводе из бинарной записи в целое число
self.OUTPUT_BYTES	3	6	В сколько байтов умещается мантисса (нужно для вывода числа)

- self.sign — знак: 0(+) или 1(–)
- self.denormilized — является ли число денормализированным (True/False) (если это вообще число, а не бесконечность или NaN)
- self.type — один из четырёх типов числа: Types_USUAL(0) — число , Types_PLUS_INF(1) — плюс бесконечность, Types_MINUS_INF(2) — минус бесконечность, или Types_NaN(3) — Not a Number)
- self.exp — целое число — значение эксоненты (если число денормализованное то всё равно это self.MIN_EXP (-14 для h и -126 для f)).
- self.mant — мантисса — list из self.MANT_LEN ноликов и единичек
- self.rounding — тип округления (целое число от 0 до 3)
- self.integer — целое число: если у нас нормальное(неденормализованное) число, то числу self.integer соответствует бинарная запись из (1 + self.MANT_LEN) битов: [1 self.mant], если денормализованное, то числу self.integer соответствует бинарная запись [0 self.mant].

Числа с плавающей точкой имеют следующие методы:

Метод	Описание
__str__()	Вывод числа в шестнадцатеричной показательной форме, со степенью в десятичном представлении
make_plus_inf()	Выдаёт число +INF
make_neg_inf()	Выдаёт число –INF
make_inf_with_sign(sign)	Выдаёт число бесконечность со знаком sign
make_NaN()	Выдаёт NaN
make_null()	Выдаёт число 0 (+0)
make_null_with_sign(sign)	Выдаёт число (+0) или (-0) в зависимости от знака sign
get_opposite()	Выдаёт противоположное число (равное по модулю, противоположное по знаку)
increase_num()	Выдаёт следующее число (то есть наименьшее число, большее данного, которое можно записать в этом же типе чисел с плавающей точкой)
decrease_num()	Выдаёт предыдущее число (то есть наибольшее число, меньшее данного, которое можно записать в этом же типе чисел с плавающей точкой)
rounding_result(sign, exponent, code)	По данным sign (0 или 1), exponent (целое число от self.MIN_EXP - 1 (случай, когда число денормализованное) до self.MAX_EXP) и списку code из 0 и 1 длины больше self.MANT_LEN во соответствии с правилом округления оставляем в code последние self.MANT_LEN значащих бит и получаем число с плавающей точкой.
rounding_result(sign, exponent, code)	По данным sign (0 или 1), exponent (целое число от self.MIN_EXP - 1 (случай, когда число денормализованное) до self.MAX_EXP) и списку code из 0 и 1 длины больше self.MANT_LEN во соответствии с правилом округления оставляем в code первые self.MANT_LEN значащих бит и получаем число с плавающей точкой.
make_ans_by_sign_exp_and_code(sign, exp, code)	По данным знаку sign (0 или 1), экспоненте exp и списку code из 0 и 1 длины или **равной** self.MANT_LEN во соответствии с правилом округления оставляем в code первые self.MANT_LEN значащих бит и получаем число с плавающей точкой.
ans_if_other_is_too_small (other_sign)	Эта функция нужна в умножениии и делении. Если произведение (частное) двух ненулевых чисел слишком мало по модулю, то нужно выдать либо 0, либо минимальное по модулю положительное ли отрицательное (в зависимости от знака) числос плавающей точкой
biggest_positive()	Случай, когда результат какой-то операции больше, чем наибольшее положительное число. Если округление к $+\infty$, выводим inf, иначе выводим наибольшее положительное число.
biggest_negative()	Случай, когда результат какой-то операции меньше, чем наибольшее по модулю отрицательное число. Если округление к $-\infty$, выводим -inf, иначе выводим наибольшее по модулю отрицательное число.
biggest$_with_sign(sign)$	biggest_positive() или biggest_negative() в зависимости от знака *sign* (если он 0, то вызываем первую функцию, если 1, то вторую).

__add__(other)	Складываем self + other. Тут рассматривается несколько случаев: 1. Складываем два денормализованных числа. Тут всё просто — взять и сложить self.integer и other.integer. Если сумма будет хотя бы $2^{\text{self.MANT\_LEN}}$, то получится номализованное число (с экспонентой self.MIN_EXP). Иначе получится снова денормализованное число. 2. Нормализованное с денормализованным. Если у нормализованного экспонента достаточно большая (хотя бы self.MANT_LEN + 2 + self.MIN_EXP), то ответ - либо то же нормализованное число, либо предыдущее или следующии (в зависимости от округления). Иначе приводим оба числа к виду $2^{\text{self.MIN\_EXP - self.MANT\_LEN}} \cdot$ x где x - целое, складываем и переводим обратно в число с плавающей точкой. 3. Два нормализованных. В целом, очень похоже на предыдущий пункт. Если складываем числа у которых экспоненты сильно различаются, то их сложить — это то же самое, что взять большее из них по модулю и, возможно (в зависимоти от округления), поменять на следующее или предыдущее (методом increase_num() или decrease_num() соответственно). Иначе оба числа приводим к меньшей степени s (в виде $2^{\text{s}} \cdot$ x где x - целое), складываеми и переводим обратно в число с плавающей точкой.
__sub__(other)	Вычитаем self − other. Это то же самое, что self + other.get_opposite()
__mul__(other)	Умножаем self · other. Переводим оба числа к виду $2^k \cdot$ 1.[self.H_MANT_LEN ноликов и единичек]: $2^{k_1} \cdot x_1$, $2^{k_2} \cdot x_2$. Их произведение — число $2^{k_1+k_2} \cdot (x_1 \cdot x_2)$. $x_1 \cdot x_2$ считаем как произведение соответсвующих целых. Далее полученное $2^{k_1+k_2} \cdot (x_1 \cdot x_2)$ аккуратно переводим в число с плавающей точкой: у нас экспонента может меняться ещё, когда $x_1 \cdot x_2$ больше 2 (но меньше 4, так как это произведение двух чисел от 1 до 2). Тогда экспоненту нужно увеличить на 1. В $x_1 \cdot x_2$ окажется больше битов, чем в мантиссе, поэтому последние нужно отбросить и, возможно, прибавить единицу в зависимости от знака произведения и типа округления.

__truediv__(other)	Делим self / other. Переводим оба числа к виду $2^k \cdot$ 1.[self.H_MANT_LEN ноликов и единичек]: $2^{k_1} \cdot x_1$, $2^{k_2} \cdot x_2$. Их отношение — число $2^{k_1-k_2} \cdot (x_1/x_2)$. Отношение x_1/x_2 считаем как частное $x_1 \cdot 2^{\text{self.MANT\_LEN + 1}}/x_2$ (сперва переведя x_1, x_2 в целые) и его переводим в вид 1.[хотя бы self.MANT_LEN ноликов и единичек] (возможно, для этого придётся домножать/делить его на два и от этого надо менять экспоненту). В итоге полученное $2^? \cdot$ 1.[сколько-то ноликов и единичек] аккуратно переводим в число с плавающей точкой.

В операциях $+, -, \cdot, /$ ещё отдельно рассматривается куча частных слуаев, когда один из аргументов или ответов NaN, какая-то бесконечность ли 0. Все эти случаи отдельно рассматриваются в коде, но я не вижу смысла тут их разбирать. Так же в коде есть комментарии, которые показывают, что происходит в некоторых строчках кода.

4.6 tester.sh

Тут есть bash-скрипт для дебага и прогона всех тестов. В нём есть сколько-то тестов, которые я сам составлял и ручками проверял, и потом тестировал на этих тестах код. Тут есть несколько наборов тестов:

- ☺ basic_tests_for_add_h — свои тесты для суммы чисел с плавающей точкой
- ☺ testsA — также тесты на сложение чисел с плавающей точкой — они писались так, чтобы покрывали все if-ы моей программы. Для каждого теста еще написано, к какому типу он относится (A и какое-то натуральное число). В самом коде есть места с комментариями вида # TESTS An, где n — какое-то натуральное число. Это значит, что специально для тестирования конкретно этой части кода придуманы тесты типа An.
- ☺ testsB — тесты на умножение чисел с плавающей точкой — абсолютно аналогичная ситуация.
- ☺ testsC — тесты на деление чисел с плавающей точкой
- ☺ testsO — тесты на вывод
- ☺ testsN — новые тесты для деления чисел с фиксированной точкой.

При запуске программы ошибок не возникает (всегда когда ожидаемый ответ не совпадает с реальным, программа это пишет):

```
andrey@andrey-ASUS:~/PycharmProjects/pythonProject1$ ./tester.sh
Testing first_tests_for_summ_h...
Testing tests_for_summ_h for all ifs...
Testing tests_for_mul_h...
Testing tests_for_div_h...
Testing tests_for_output_and_some_other...
Testing New_tests...
Check finished
```

PS. После дедлайна ещё добавились тесты. Обработан случай с ± 0 при сложениии чисел с плавающей точкой. Если мы складываем два –0 (с любым округлением), а также если складываем два числа a и $-a$ и округляем к $-\infty$, получаем –0, при других округлениях — $+0$. Обработан случай, когда мы получаем самое большое по модулю число: раньше, когда мы что-нибудь складывали и получали слишком большое, мы говорили, что получится $+infty$ всегда, это было неправильно. Теперь, если мы сложили и получили что-то большее, чем наибольшее положительное число, мы получим наибольшее положительное число, если округление не к $+\infty$, аналогично с отрицательными числами. Пофиксил ещё некоторые баги в умножении и делении чисел с фиксированной точкой.